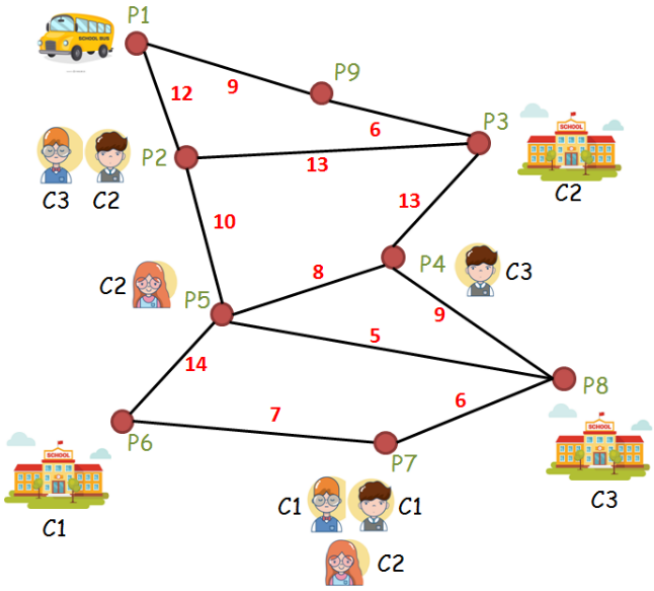

Heurística y Optimización

Práctica 2: Satisfacción de Restricciones y Búsqueda Heurística

Antonio Viñuela Pérez – 100383461
Fernando Pérez Lozano – 100383462

Contenidos

1. Introducción al documento

En este documento describiremos las tareas que hemos resuelto, y analizaremos tanto el método utilizado para resolverlas como los resultados que hemos obtenido.

La práctica consta de dos partes:

- Parte 1: Satisfacción de restricciones

Se plantea un problema de compatibilidad de horarios entre las asignaturas de un colegio, así como de asignación de profesores a dichas asignaturas. Esta parte se modelizará y solucionará usando la librería *python-constraint*.

- Parte 2: Búsqueda Heurística

Se debe buscar la ruta óptima para un autobús que recoge niños de diferentes colegios en diferentes paradas, disponiendo de la posición inicial tanto de los niños como del bus y de los colegios, así como del estado final: todos los niños entregados y el bus de vuelta en su posición de origen. También se conoce el coste de moverse entre las diferentes paradas, representadas con un grafo. Para esta parte, tendremos que implementar el algoritmo de A* en el lenguaje de programación que deseemos (hemos elegido Java).

Al final del documento expondremos nuestras conclusiones acerca de la práctica, así como las dificultades con las que nos hemos encontrado en la elaboración de esta.

2. Descripción de los modelos

Parte 1

Para resolver este problema, hemos hecho uso de la librería *python-constraint*:

1. **Variables:**

Hemos declarado dos sets de variables: asignaturas y profesores.

a) Asignaturas**:** Naturales, Sociales, Lengua, Matemáticas, Ingles, Educación Física

Dado que todas las asignaturas deben ser impartidas dos veces a la semana excepto Educación Física, hemos modelado el conjunto de variables como:

$$X = (N1, N2, S1, S2, L1, L2, M1, M2, I1, I2, F)$$

b) Profesores: Lucía (C), Andrea (A), Juan (U)

Todos los profesores deben impartir dos horas de clase. Por ello, el conjunto de variables será:

$$Y = (C1, C2, A1, A2, U1, U2)$$

2. **Dominios:**

Hemos utilizado dos conjuntos de dominios, Horarios y Asignaturas, relacionados con las variables Asignaturas y Profesores respectivamente:

a) Horarios: L1, L2, L3, M1, M2, M3, X1, X2, X3, J1, J2

Los dominios del set de asignaturas serán los horarios, pues cada variable debe asignarse a un día y hora específico. Los hemos modelado utilizando números enteros, para facilitar la implementación de restricciones. Hemos seguido la siguiente leyenda:

```
  0     1     2     3     4     5     6     7     8     9    10    11    12    13    14
[None, 'l1', 'l2', 'l3', None, 'm1', 'm2', 'm3', None, 'x1', 'x2', 'x3', None, 'j1', 'j2']
```

Algunas restricciones limitan directamente el dominio inicial de ciertas variables. Esto será desarrollado en la sección que describe las restricciones.

b) Asignaturas: N, S, L, M, I, F

Los dominios del set de profesores serán las asignaturas, pues cada profesor debe ser asignado a dos asignaturas. En este caso no hemos necesitado modelar el dominio como números enteros, pues no fue necesario para ninguna restricción.

3. **Restricciones:**

En el enunciado hay un total de 8 restricciones.

a) La duración de cada una de las clases es de 1 hora, en la que solo se puede impartir una única materia.

Esta restricción nos ayudó a decidir el formato del dominio de los horarios, dividiendo cada día en tres partes (primera, segunda y tercera hora) excepto el jueves, que solo tiene dos horas.

b) Para todas las materias se deben impartir 2 horas semanales, excepto para Educación Física que solo tiene asignada 1 hora semanal.

Esta restricción nos ayudó a decidir el formato del conjunto de variables de las asignaturas, dividiendo cada una en dos partes (una por cada hora a impartir) excepto Educación Física, que solo es impartida una hora a la semana.

c) Las 2 horas dedicadas a cada materia podrían impartirse de forma no consecutiva, e incluso en días diferentes, excepto las 2 horas dedicadas a Ciencias de la Naturaleza que sí se deben impartir de forma consecutiva el mismo día.

Primera restricción que implementamos con addConstraint. Creamos una función comprobando N1 y N2 sean consecutivos, que esencialmente el significado de la restricción. Para ello, forzamos que la resta de los dominios de N1 y N2 sea exactamente 1 (valor absoluto, de modo que no importe si N2 va antes que N1 o viceversa), lo cual quiere decir que se imparten consecutivamente y en el mismo día.

Para esta restricción decidimos dejar una diferencia de 1 en los valores del dominio de las asignaturas: por ejemplo, L3 es 3 y M1 es 5. De esta forma, nunca se confundirán horas consecutivas pero en días diferentes con horas de verdad consecutivas.

d) La materia de Matemáticas no puede impartirse el mismo día que Ciencias de la Naturaleza e Inglés.

Para esta restricción, usamos el array *indiceDias* en el que guardamos en cada correspondiente índice del dominio Horarios su significado en caracteres (indiceDias[1]=L1, indiceDias[6]=M2, etc), de acuerdo a la leyenda expuesta en la descripción del dominio.

Usando este array, accedemos al primer caracter de cada posible combinación de dominios forzándolos a ser distintos. De este modo, evitamos que las asignaturas se impartan el mismo día. Usamos este constraint para las tuplas (M,N) y (M,I) en todas sus correspondientes horas (M1, M2, N1…).

e) La materia de Matemáticas debe impartirse en las primeras horas, y la de Ciencias Sociales en las últimas

Esta restricción la aplicamos restringiendo directamente, al declararlos, los posibles valores de los dominios de M1 y M2 a 1, 5, 9 y 13 (primeras horas de cada día). De forma similar, restringimos los dominios de S1 y S2 a 3, 7, 11 y 14 (últimas horas de cada día).

f) Cada profesor debe impartir 2 materias, que son diferentes a las de sus compañeros.

Esencialmente, esta restricción implica que cada instancia de cada profesor (C1, C2, A1, …, U2) debe tener un dominio único y diferente de los dominios de los demás. Como las asignaturas también deben tener dominios únicos, simplemente añadimos un AllDifferentConstraint para todas las variables del problema. Esto lo hacemos al final del programa, para no limitar totalmente los dominios antes de hacer todas las comprobaciones.

g) Lucía solo se encargará de Ciencias Sociales, si Andrea se encarga de Educación Física.

Hemos entendido esta restricción como que, en el caso de que Lucía (C1 o C2) tenga como dominio Sociales (S), necesariamente Andrea (A1o A2) tendrá que ser asignada a Educación Física. En caso de que el dominio de Lucía no incluya Sociales, no se hará ninguna comprobación adicional.

h) Juan no quiere encargarse de Ciencias de la Naturaleza o de Ciencias Sociales, si algunas de sus horas se imparten a primera hora los lunes y jueves.

Para esta restricción, hemos implementado una función que reciba como argumento todas las posibles combinaciones de Juan (U1 o U2) con Sociales (S1 o S2) o con Naturales (N1 o N2). Esta función devolverá siempre True excepto en el caso de que la asignatura recibida tenga como dominio 1 o 13 (lunes a primera y jueves a primera, respectivamente) y que Juan tenga asignada dicha asignatura.

4. Solución:

Al evaluar una posible solución, usando el método getSolution, obtenemos lo siguiente:

Asignaturas: N1: 14, N2: 13 → jueves a segunda y jueves a primera

S1: 11, S2: 7 → miércoles a tercera y martes a tercera

L1: 6, L2: 1 → miércoles a segunda y lunes a primera

M1: 9, M2: 5 → miércoles a primera y martes a primera

I1: 3, I2: 2 → lunes a tercera y lunes a segunda

F: 10 → miércoles a segunda

Profesores: Lucia → C1: Lengua, C2: Naturales

Andrea → A1: Mates, A2: Sociales

Juan → U1: Educación Física, U2: Inglés

Al pasar a evaluar todas las soluciones posibles usando getSolutions, el programa nunca parece terminar de ejecutarse. Aproximadamente a los 20 minutos, recibimos un MemoryError. Esto nos parece extraño, puesto que a priori, y tras muchas revisiones, no parece que haya ningún problema con las restricciones que hiciera que tomara tanto tiempo e iteraciones y además habiendo probado el programa en diferentes ordenadores con diferentes capacidades de RAM.

```
Traceback (most recent call last):
  File "CSPScheduling.py", line 113, in <module>
    print(problem.getSolutions())
  File "C:\Python27\lib\site-packages\constraint\__init__.py", line 271, in getSolutions
    return self._solver.getSolutions(domains, constraints, vconstraints)
  File "C:\Python27\lib\site-packages\constraint\__init__.py", line 567, in getSolutions
    return list(self.getSolutionIter(domains, constraints, vconstraints))
MemoryError
```

Error recibido al ejecutar el programa en un ordenador con 16gb de RAM.

Como casos de prueba, lo primero que podríamos hacer sería eliminar la restricción que hace que todos los dominios sean diferentes (AllDifferentConstraint), que debería ser la más restrictiva. Esto haría que el algoritmo expandiese muchos más nodos y generase muchas posibles soluciones adicionales.

Otra prueba que podría hacerse es añadir dos asignaturas con dos horas semanales cada una, sin restricciones y ampliar el horario de clases añadiendo jueves a tercera y viernes a primera y segunda. Esto añadiría más carga al problema, y haría que el solucionador de nuevo generase muchos más nodos, pues esas dos asignaturas podrían asignarse a cualquier horario. Además, se añadirían evaluaciones a los dominios de las asignaturas originales, pues tendrían cuatro nuevas posibilidades de asignación (las cuatro nuevas horas).

Al contrario, podríamos eliminar una asignatura de dos horas semanales, como Lengua, y eliminar también dos horas de la semana. Esto generaría menos nodos y consecuentemente tardaría menos tiempo en generar un conjunto de soluciones.

Como última prueba, se podría eliminar un profesor, por ejemplo Juan, y dos asignaturas. Esto de nuevo reduciría la cantidad de nodos expandidos por el programa, reduciendo así el tiempo que tomaría para generar todas las posibles soluciones.

Parte 2

Hemos decidido usar Java para implementar y solucionar este problema.

1. Estados:

Para definir y representar cada estado del problema, hemos usado un formato de estado que incluye un array de objetos Parada de tamaño igual al número de parada, más una variable de tipo Guagua, que representará el estado del bus (en Canarias, autobús se dice guagua así que decidimos ser fieles a nuestra tierra).

- **Parada**: Cada objeto Parada contendrá un array de enteros, que tendrá en cada correspondiente índice el número de niños de cada colegio (por ejemplo, en el índice 0 estarán representados los niños del colegio 1 esperando a ser recogido en esa parada), seguido de otro array de booleans que representa en cada índice la existencia o no en esa parada del correspondiente colegio (por ejemplo, el array [F,F,T,F] indica que hay cuatro colegios en el problema y que en esa parada se encuentra el colegio 3). Ambos arrays serán inicializados con un tamaño igual al número de colegios en el problema.
- **Guagua**: Cada objeto Guagua se compone de dos elementos: un array de enteros de tamaño igual al número de colegios en el problema, representando en cada índice el número de niños del correspondiente colegio que se encuentran en ese momento en la guagua; y un entero representando la parada en la que se encuentra la guagua en el momento representado.

2. Acciones:

Para decidir qué opciones expandir a partir de un determinado estado, hemos definido tres posibles acciones:

- **Mover (s, t)** → mueve la guagua de la parada s a la parada t.
  - **Precondiciones:**
    - coste(s,t) > 0 → Existe coste positivo entre ambas paradas. Esto implica que las paradas son adyacentes.
    - Estado.guagua.posicion = s → La guagua se encuentra en la parada s.
  - **Efectos:**
    - Estado.guagua.posicion = t → Actualizar posición de la guagua.

- **Recoger (P, c)** → recoge un alumno del colegio c de la parada P.
  - **Precondiciones:**

  - Estado.guagua.posicion = P → La guagua se encuentra en la parada P.
  - Estado.paradas[P].alumnos[c] > 0 → hay algún niño del colegio c en la parada P.
  - Capacidad actual del bus > 0 → queda espacio en la guagua.
- **Efectos:**
  - Estado.paradas[P].alumnos[c] – 1 → se quita el alumno de la parada P.
  - Estado.guagua.alumnos[c] + 1 → se añade el alumno a la guagua

- **Entregar (P, c)** → entrega un alumno del colegio c a la parada P.
  - **Precondiciones:**
    - Estado.paradas[P].colegios[c] == true → en la parada P está el colegio c.
    - Estado.guagua.alumnos[c] > 0 → hay algún alumno del colegio c en la guagua.
  - **Efectos:**
    - Estado.guagua.alumnos[c] – 1 → se quita el alumno de la guagua.

3. Función heurística:

Hemos definido dos funciones heurísticas informadas y admisibles, una más precisa que la otra:

- **Función heurística 1: *<heuristica1>***

  La heurística más completa que se nos ha ocurrido. Será el sumatorio de todos los niños de cada colegio en el tablero (en parada o en bus) por la distancia hasta su correspondiente colegio. Para esto, usaremos el algoritmo de Floyd-Warshall.

  La fórmula será:

$$hFW = \sum_{1}^{p} hp + hb$$

  donde:

  - p es el número de paradas,
  - hp es el sumatorio del coste de llevar a todos los niños de cada parada p hasta su colegio:
    - n = número de colegios,
    - x = colegio que se está comprobando
    - Cx = alumnos del colegio x en la parada
    - Dist(P, x) = distancia entre la parada P y el colegio x

$$hp = \sum_{x=1}^{n} Cx * dist\ (P, x)$$

  - hb es el coste de llevar a todos los niños del bus hasta sus colegios:
    - n = número de colegios

- x = colegio que se está comprobando
- Cx = alumnos del colegio x en el bus
- Dist(P, x) = distancia entre el bus B y el colegio x

$$hb = \sum_{x=1}^{n} Cx * dist\ (B, x)$$

- **Función heurística 2: *<heuristica2>***

  Esta será la heurística menos precisa. Usaremos simplemente el número de niños pendientes de entregar, ya se encuentren en la guagua o esperando en una parada.

La utilización de la primera heurística debería hacer que el programa generase menos nodos y tomase menos tiempo, puesto que es mucho más precisa y completa que la segunda. Sin embargo, por el error que sufre nuestro programa (que explicamos en el apartado 5), esto no siempre es así. Al generarse en algún momento que no logramos identificar un estado final y añadirse a la lista de estados cerrados, esto es impredecible.

4. Implementación

Hemos implementado el problema de la siguiente forma:

1. Empezamos creando el estado inicial a partir del input y el estado final a partir del estado inicial: debe acabar en la parada inicial pero sin niños en el tablero, tanto en las paradas como en la guagua.
2. Creamos las listas, usando ArrayLists de Estados, para los estados que se encuentren abiertos y los que se encuentren cerrados.
3. Creamos la lista de sucesores, que se renovará con cada iteración del algoritmo. Creamos también la variable booleana *exito*, que será evaluada al final del problema para saber si se ha encontrado o no una solución.
4. Empieza el bucle de A*. Este bucle se ejecutará mientras queden estados por expandir en la lista de abiertos, y será detenido en caso de encontrar el estado final. Se procederá a evaluar el primer estado en la lista de abiertos (es decir, el que tiene menor función de coste)
5. Si el estado evaluado no es el final, se expandirá generando sus sucesores y se retirará de la lista de abiertos. Además, se añadirá a la lista de estados cerrados.
   En caso de que sí sea el estado final, se hará True la variable *exito* y se acabará el bucle de A*.
6. Se evalúan todos los posibles estados sucesores que se pueden crear. Para esto, se comprueba la posibilidad de ejecutar cada una de las posibles acciones para el estado actual, usando un bucle for para cada una de las tres acciones (mover, recoger y entregar).
7. Se comprueba cada estado sucesor (s) generado:
   a. Si s no está en la lista abierta ni en la cerrada, se inserta en la lista provisional de estados sucesores.
   b. Si s está en la lista abierta pero su nueva función de coste es mejor que la anterior, se borra el estado anterior de la lista abierta y el actual se añade a la lista de sucesores.
   c. Si s está en la lista cerrada se ignora, pues significa que ya ha sido expandido previamente

8. Se ordena la lista provisional de sucesores en base a las funciones de coste de cada estado (de menor a mayor) y luego esta lista se mergea con la lista de estados abiertos, introduciendo los nuevos estados de nuevo en base a sus funciones de coste. Esto requiere una complejidad mucho menos que si tuviéramos que ordenar la lista de estados abiertos en cada iteración. Tras estas operaciones, se limpia la lista de estados sucesores.
9. Volver al paso 4.

Tras completar todos los pasos (es decir, que el bucle se termine por alcanzar una solución o porque la lista de estados abiertos se vacía) se evaluará la variable *éxito* y se determinará si se ha llegado o no a una solución.

5. Solución

No conseguimos que el algoritmo devuelva la solución óptima para todos los inputs. Hemos encontrado ciertos inputs para los que sí funciona, como que los niños del problema se encuentren en la parada inicial o que no haya niños en el tablero.

Tenemos el error identificado: hay alguna iteración del algoritmo que incluye al estado final en la lista de nodos cerrados. Esto no tiene sentido, pues lo que hace nuestro A* es inmediatamente después de tomar un estado de la lista de estados abiertos, comprobar si este es el estado final. Por ello, nunca debería ser capaz de añadir el estado final a la lista cerrada.

Debido a este error, el algoritmo expande estados continuamente hasta que se encuentra que la lista de estados abiertos está vacía. Obviamente, dado que el estado final se incluye en algún momento en la lista cerrada, nunca es incluido en la lista abierta. Por ello, nunca consigue llegar a él y el algoritmo termina cuando ha expandido todos los nodos de la lista abierta.

Después de mucho rato intentando encontrar el error, hemos decidido dejarlo así, continuar con el resto de la práctica, y volver a intentarlo en el caso de que nos sobrara tiempo (que no ha sucedido).

3. Conclusiones

Aunque no hayamos conseguido implementar correctamente A* en Java, sí que hemos llegado a comprender perfectamente el funcionamiento del mismo, por lo que creemos que esta práctica nos ha ayudado bastante a profundizar y asimilar el contenido de la materia. Esto nos ayudará tanto de cara al examen como de cara al futuro, en el que esperamos seguir estudiando el campo de la heurística y optimización.

Está claro que, sobre todo por la segunda parte, no nos ha resultado una práctica fácil — en tal medida que no hemos sido capaces de resolverla del todo. Realmente lo hemos intentado, pero ha llegado a desesperarnos y al ver que se nos iba echando el tiempo encima lo hemos dejado como estaba.

Sin embargo, estamos contentos con el resultado obtenido en la parte 1. Aunque por alguna razón genera tantas posibles combinaciones de dominios que abruma al ordenador, creemos que está correctamente implementado: todas las soluciones individuales que hemos comprobado con diferentes casos de prueba han sido correctas. Ha sido interesante comprobar que un problema que a priori parece relativamente sencillo de resolver es capaz de devolver tantos posibles resultados.